# THE UNIVERSITY OF AUCKLAND

**First Semester, 2005**

**City Campus**

**Computer Science**

**Language Implementation**

(Time allowed TWO hours)

| |
|---|
| FAMILY NAME: |
| PERSONAL NAMES: |
| STUDENT ID NUMBER: |
| LOGIN NAME: |
| SIGNATURE: |

This Examination is out of 100 Marks. Attempt **ALL** questions. Write your answers in the spaces provided in this question and answer booklet. Do not remove the staples from the question and answer booklet. However, you may detach and remove the staples from the appendices.

| | | |
|---|---|---|
| 1 | | 24 |
| 2 | | 14 |
| 3 | | 15 |
| 4 | | 15 |
| 5 | | 16 |
| 6 | | 16 |
| Total | | 100 |

Continued ...

Print Name ____________________________________________

## 1. Bottom Up LALR(1) Parsing [24 Marks]

Consider the CUP grammar in the **Appendix For Question 1**. Note that some rules are left recursive, while other rules are right recursive. Also note the rule for "ConcatExpr" that expands to empty.

(a) Using the information provided in the appendix, perform a shift-reduce LALR(1) parse of the input

```
a[bc]|d
```

Assume "`a` ", "`b` ", "`c` ", "`d`", match `CHAR`, and "`[`", "`]`" and "`|`" match `LEFTSQ`, `RIGHTSQ` and `OR`, respectively.

Stack

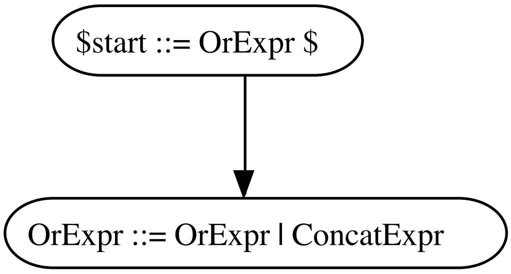
| | | | | |
|---|---|---|---|---|
| $0 | | | | |
| $0 | CE 2 | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | | | | |
| $0 | OE 1 | $18 | | |
| $0 | $start -1 | | | |

| Token | Action | |
|---|---|---|
| CHAR a | Reduce | CE ::= |
| | Shift | CHAR 5 |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | Reduce | $START ::= OE $ |
| | Accept | |

(14 marks)

Print Name ____________________

(b) Draw the parse tree corresponding to the grammar rules used to parse this input




(4 marks)

Continued ...

Print Name ______________________________

(c) State 2 is

```
lalr_state [2]: {
     [SimpleExpr ::= (*) LEFTSQ ElementList RIGHTSQ ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [OrExpr ::= ConcatExpr (*) , {EOF OR RIGHT }]
     [SimpleExpr ::= (*) LEFT OrExpr RIGHT ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [ConcatExpr ::= ConcatExpr (*) SimpleExpr ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [SimpleExpr ::= (*) CHAR ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
}
```

Derive the set of items of State 6 = GoTo( State 2, LEFTSQ ). Remember to take the closure to get the full set of items, and remember to compute the follow symbols.

(6 marks)

Print Name ______________________________________________

**2. Write a grammar for variable declarations** **[14 Marks]**

A (simplified) Java variable declaration is composed of a type, followed by a "," separated list of declarators, then a ";". A type is either a primitive type ("int", "char", etc), identifier (e.g., "String"), or an array type (a type followed by "[]"s, e.g., "int[]", "String[][]"). A declarator can be either uninitialised (just an identifier) or initialised ("identifier = initialiser"). An initialiser can be either an expression, or an array initialiser. An array initialiser is an optional "," separated list of initialisers enclosed in "{...}"s.

For example, the following represent variable declarations:

```
int a, b, d = 3, e = 4, f;
int[] g = { 1, 2, 3 }, h = new int[ 3 ], i;
int[][] j = { { 1, 2, 3 }, new int[ 3 ], { 4, 5, 6 } };
String[]  k = { "yes", "no", "maybe" };
```

Write a grammar for variable declarations **in general**. You do not have to define what a primitive type or expression is (and note that array constructors such as "new int[ 3 ]" are expressions). Unlike Java, you should not allow modifiers, "[]"s after the identifier being declared, or an additional "," after the last initialiser in an array initialiser. You do not have to write actions.

Print Name ______________________________

(14 marks)

Continued ...

Print Name ______________________________

**3. Interpretation** **[15 Marks]**

Consider the INTERP9/BHP language of Assignment 2.

(a) Describe the general structure of a runtime environment.

(1 mark)

(b) Explain how var parameters can be implemented using this representation.

(2 marks)

Continued ...

Print Name ____________________________________________

(c) Explain how a runtime environment can be used to represent an array. Give an example of a program with

- A function that takes a list of values as parameters, and creates an array with those values as elements.
- A function that takes an array as a parameter, and loops printing the values of the elements.
- Global code to invoke the functions to create the array and print out its elements.

(6 marks)

(d) Draw a diagram showing the runtime environments that would be generated by the following program, at the time when the function f is being invoked.

```
function f( $a, $b, &$c )
     begin
          //   show at this point
     end
$x = happy;
$y = x;
$z = $x;
f( sad, $x, &$x, $y );
```

Print Name ____________________

(6 marks)

Continued ...

Print Name ______________________________________________

**4. Show the run time stack for an INTERP7 program [15 Marks]**

Use the program written in the Chapter 8 INTERP7 language in the **Appendix For Question 4**.

Complete the drawing of the data structure built for the global variables "`source1`", "`source2`" and "`dest`".

Display the stack frames (activation records) for all methods in the process of being invoked when the maximum level of nesting of method invocations occurs when the statement

```
merge( 0, source1, source2, dest );
```

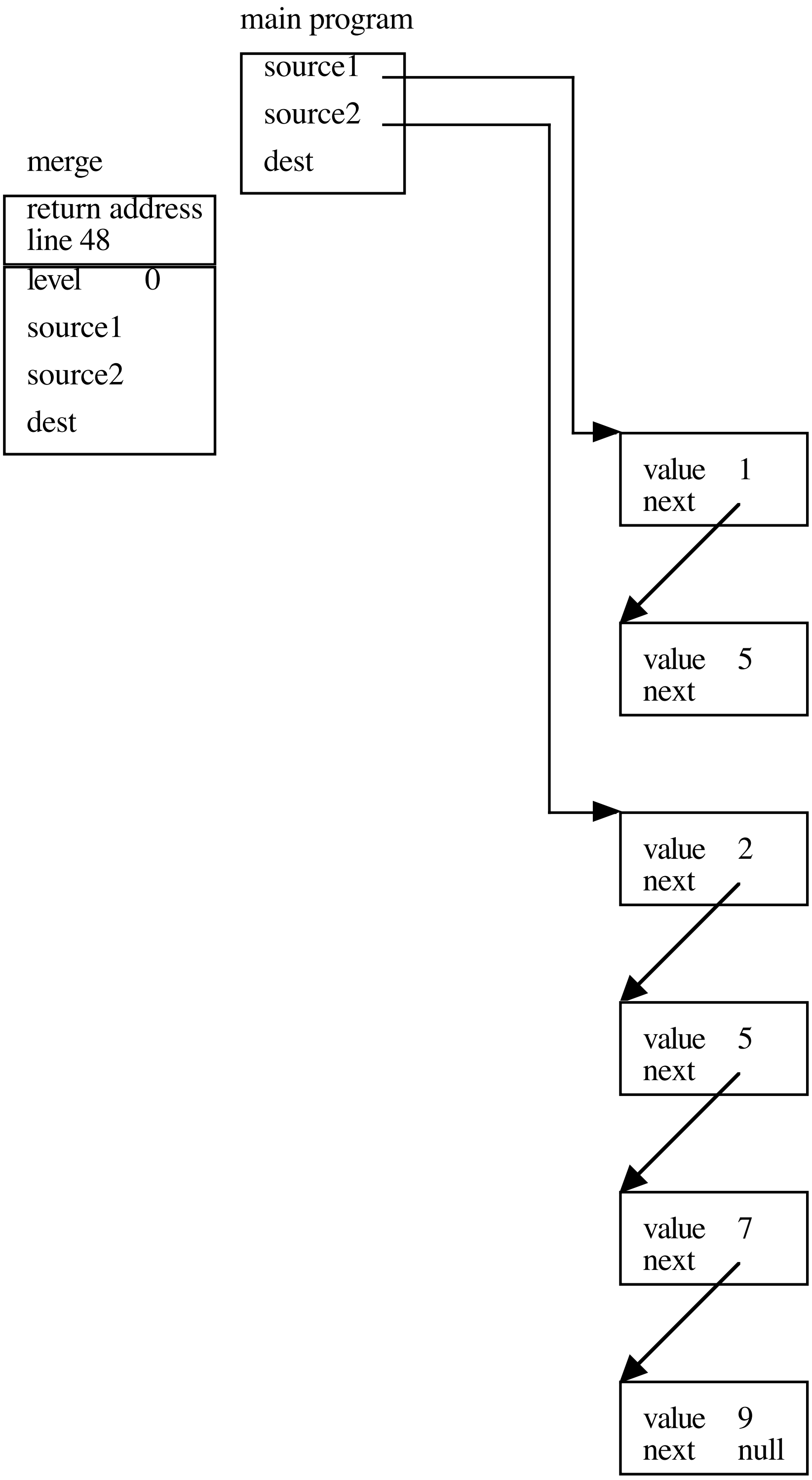
on line 47 is invoked, and the process is almost ready to return. At this stage the process should be executing the method "`merge`" at line 33.

Indicate the appropriate values for each stack frame (activation record) you draw. The line numbers on the left-hand side of the program should be used to represent the return address. Draw appropriate arrows for the var parameters, and pointers to objects. For var parameters, make sure you indicate the exact field pointed to in an object very clearly. Represent `List` nodes as shown in the sample entries.

Also indicate the output generated by the complete execution of the program.

| Output generated: |
|---|
| (3 marks) |

Print Name ____________________________________________



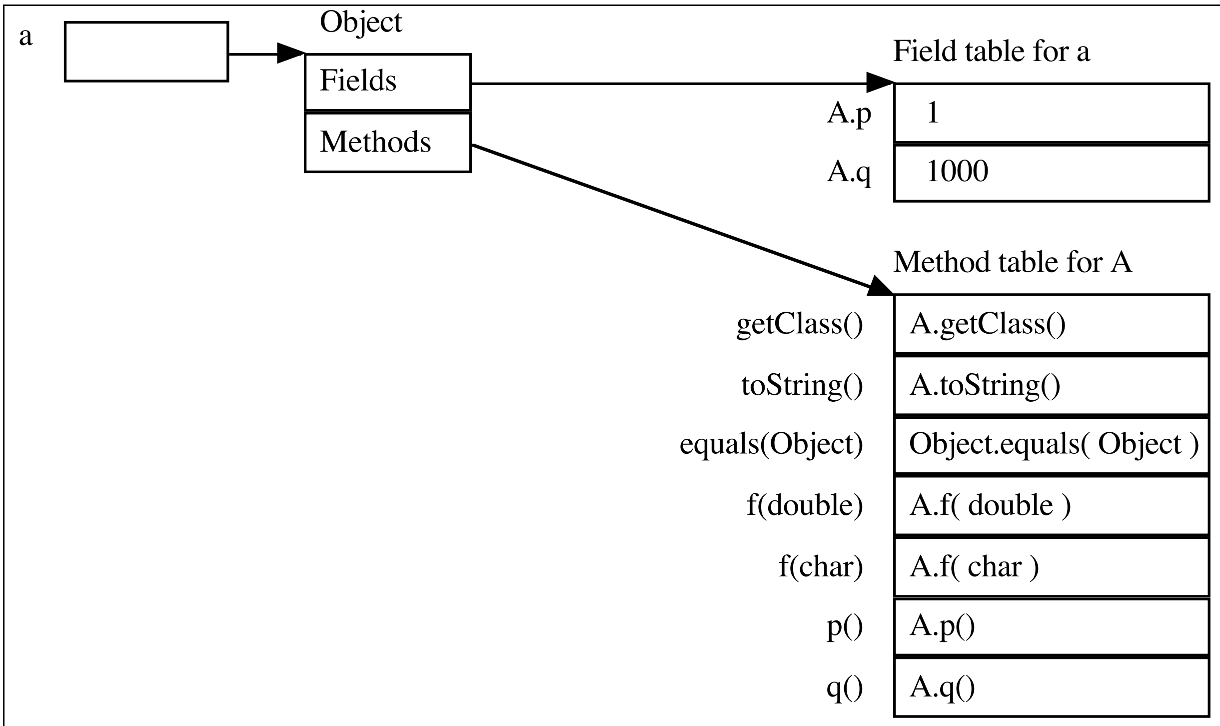

(12 marks)

Print Name ______________________________________________

**5. Implementation of object oriented languages** **[16 Marks]**

Use the Java program in the **Appendix For Question 5**.

(a) Draw a diagram showing the data structures (object, field table, method table, etc) created for the variables `a`, `b1`, `b2`, `c`, within the method `Main.main`. Shared data structures should be drawn only once.



Print Name ______________________________

(11 marks)

(b) Indicate the output generated by the method Main.main.

```
A            a.p  = 1      a.q  = 1000
             b1.p =        b1.q =
             b2.p =        b2.q =
             c.p  =        c.q  =
             b1.p() =      b1.q() =
             b2.p() =      b2.q() =
b1.f( 'A' ) =
b2.f( 'A' ) =
b1.f( 65 ) =
b2.f( 65 ) =
```

(5 marks)

Continued ...

Print Name ______________________________________________

**6. Code generation** **[16 Marks]**

Consider the program written in the assignment 3 OBJECT7 language in the **Appendix For Question 6**.

(a) Explain why the lines

```
y = a;
z = list[ i ];
```

are legal in the OBJECT7 language.

(2 marks)

(b) Indicate the code likely to be generated to implement the following statements

**Notes:**

**An appendix is provided with common Alpha instructions.**

**Addresses are represented using 8 bytes on the Alpha.**

**Each statement should be translated independently, and not make use of values left in registers from previous declarations or statements.**

```
y = a;
```

(1 mark)

```
y = p;
```

(1 mark)

Continued ...

Print Name ___________________________________________

```
x = p^;
```

(2 marks)

```
x = a[ i ];
```

(2 marks)

```
y = &a[ i ];
```

(2 marks)

Continued ...

Print Name ______________________________

```
y = b[ i ];
```

(2 marks)

```
z = list[ i ];
```

(2 marks)

```
z++;
```

(2 marks)

_______________End of Questions_______________

Continued ...

Print Name ____________________________________________

This Page is left blank for questions that overflow

Print Name ____________________________________________

This Page is left blank for questions that overflow

End of Questions, Appendices follow

## Appendix For Question 1

## Grammar

```
terminal String
        OR, LEFT, RIGHT,   LEFTSQ,   RIGHTSQ,  MINUS,    CHAR;
    //  |    (    )        [         ]         -         Any other char

non terminal OrExpr, ConcatExpr, SimpleExpr, ElementList, Element;

start with OrExpr;

OrExpr::=
        OrExpr OR ConcatExpr
    |
        ConcatExpr
    ;

ConcatExpr::=
        /* Empty*/
    |
        ConcatExpr SimpleExpr
    ;

SimpleExpr::=
        LEFTSQ ElementList RIGHTSQ
    |
        CHAR
    |
        LEFT OrExpr RIGHT
    ;

ElementList::=
        Element
    |
        Element ElementList
    ;

Element::=
        CHAR
    |
        CHAR MINUS CHAR
    ;
```

## Appendix For Question 1 Continued On Next Page ....

## Appendix For Question 1 Continued ...

## Rules

```
11: Element ::= CHAR MINUS CHAR
10: Element ::= CHAR
9: ElementList ::= Element ElementList
8: ElementList ::= Element
7: SimpleExpr ::= LEFT OrExpr RIGHT
6: SimpleExpr ::= CHAR
5: SimpleExpr ::= LEFTSQ ElementList RIGHTSQ
4: ConcatExpr ::= ConcatExpr SimpleExpr
3: ConcatExpr ::=
2: OrExpr ::= ConcatExpr
1: OrExpr ::= OrExpr OR ConcatExpr
0: $START ::= OrExpr EOF
```

## Action Table

```
From state #0
     EOF:REDUCE(3) OR:REDUCE(3) LEFT:REDUCE(3)
     LEFTSQ:REDUCE(3) CHAR:REDUCE(3)
From state #1
     EOF:SHIFT(18) OR:SHIFT(16)
From state #2
     EOF:REDUCE(2) OR:REDUCE(2) LEFT:SHIFT(3)
     RIGHT:REDUCE(2) LEFTSQ:SHIFT(6) CHAR:SHIFT(5)
From state #3
     OR:REDUCE(3) LEFT:REDUCE(3) RIGHT:REDUCE(3)
     LEFTSQ:REDUCE(3) CHAR:REDUCE(3)
From state #4
     EOF:REDUCE(4) OR:REDUCE(4) LEFT:REDUCE(4)
     RIGHT:REDUCE(4) LEFTSQ:REDUCE(4) CHAR:REDUCE(4)
From state #5
     EOF:REDUCE(6) OR:REDUCE(6) LEFT:REDUCE(6)
     RIGHT:REDUCE(6) LEFTSQ:REDUCE(6) CHAR:REDUCE(6)
From state #6
     CHAR:SHIFT(9)
From state #7
     RIGHTSQ:REDUCE(8) CHAR:SHIFT(9)
From state #8
     RIGHTSQ:SHIFT(12)
From state #9
     RIGHTSQ:REDUCE(10) MINUS:SHIFT(10) CHAR:REDUCE(10)
From state #10
     CHAR:SHIFT(11)
```

## Appendix For Question 1 Continued On Next Page ....

## Appendix For Question 1 Continued ...

```
From state #11
     RIGHTSQ:REDUCE(11) CHAR:REDUCE(11)
From state #12
     EOF:REDUCE(5) OR:REDUCE(5) LEFT:REDUCE(5)
     RIGHT:REDUCE(5) LEFTSQ:REDUCE(5) CHAR:REDUCE(5)
From state #13
     RIGHTSQ:REDUCE(9)
From state #14
     OR:SHIFT(16) RIGHT:SHIFT(15)
From state #15
     EOF:REDUCE(7) OR:REDUCE(7) LEFT:REDUCE(7)
     RIGHT:REDUCE(7) LEFTSQ:REDUCE(7) CHAR:REDUCE(7)
From state #16
     EOF:REDUCE(3) OR:REDUCE(3) LEFT:REDUCE(3)
     RIGHT:REDUCE(3) LEFTSQ:REDUCE(3) CHAR:REDUCE(3)
From state #17
     EOF:REDUCE(1) OR:REDUCE(1) LEFT:SHIFT(3)
     RIGHT:REDUCE(1) LEFTSQ:SHIFT(6) CHAR:SHIFT(5)
From state #18
     EOF:REDUCE(0)
```

## Reduce (Go To) Table

```
From state #0:
     OrExpr:GOTO(1)
     ConcatExpr:GOTO(2)
From state #1:
From state #2:
     SimpleExpr:GOTO(4)
From state #3:
     OrExpr:GOTO(14)
     ConcatExpr:GOTO(2)
From state #4:
From state #5:
From state #6:
     ElementList:GOTO(8)
     Element:GOTO(7)
From state #7:
     ElementList:GOTO(13)
     Element:GOTO(7)
From state #8:
From state #9:
From state #10:
From state #11:
From state #12:
From state #13:
From state #14:
From state #15:
From state #16:
     ConcatExpr:GOTO(17)
From state #17:
     SimpleExpr:GOTO(4)
From state #18:
```

## Appendix For Question 4

```
class List( int value; List next; );

void printList( List source; ) {
 print( "{ " );
 while ( source != null ) {
    print( source.value );
    source = source.next;
    if ( source != null )
        print( ", " );
        }
    println( " }" );
    }

 void merge( int level; List source1, source2; var List dest; ) {
    if ( source1 == null ) {
        dest = source2;
        }
    else if ( source2 == null ) {
        dest = source1;
        }
    else if ( source1.value < source2.value ) {
        dest = new List{ source1.value, null };
        merge( level + 1, source1.next, source2, dest.next );
        }
    else if ( source1.value > source2.value ) {
        dest = new List{ source2.value, null };
        merge( level + 1, source1, source2.next, dest.next );
        }
    else if ( source1.value == source2.value ) {
        dest = new List{ source1.value, null };
        merge( level + 1, source1.next, source2.next, dest.next );
        }
    // Show state at this point
    }

 List source1 =
    new List{ 1,
    new List{ 5,
        null } };
 List source2 =
    new List{ 2,
    new List{ 5,
    new List{ 7,
    new List{ 9,
        null } } } };
 List dest = null;
 merge( 0, source1, source2, dest );  // Inside this invocation
 printList( source1 );
 printList( source2 );
 printList( dest );
```

## Appendix For Question 5

```
class A {

     public int p = 1, q = 2;

     public A( int q ) { this.q = q; }
     public A() { p = 4; }

     public String toString() { return "A"; }
     public String f( double x ) { return "A.f( " + x + " )"; }
     public String f( char c ) { return "A.f( '" + c + "' )"; }
     public int p() { return p; }
     public int q() { return q; }
     }

class B extends A {

     public int p = 5, q = 6;

     public B( int p ) { this.p = p; }
     public B() {}

     public String toString() { return "B"; }
     public int q() { return q; }
     public String f( int i ) { return "B.f( " + i + " )"; }
     public String f( double x ) { return "B.f( " + x + " )"; }
     }

class C extends A {
     public C() { q = 3000; }
     }
```

**Appendix For Question 5 Continued On Next Page ....**

## Appendix For Question 5 Continued From Previous page ...

```
class Main {
     public static void main( String[] args ) {

          A a = new A( 1000 );
          B b1 = new B( 2000 );
          A b2 = b1;
          C c = new C();
          b1.q = 8;
          b2.q = 9;

          System.out.println( a  + "\ta.p  = " +  a.p + "\ta.q  = " +  a.q );
          System.out.println( b1 + "\tb1.p = " + b1.p + "\tb1.q = " + b1.q );
          System.out.println( b2 + "\tb2.p = " + b2.p + "\tb2.q = " + b2.q );
          System.out.println( c  + "\tc.p  = " +  c.p + "\tc.q  = " +  c.q );
          System.out.println();

          System.out.println( "\tb1.p() = " + b1.p() + "\tb1.q() = " + b1.q() );
          System.out.println( "\tb2.p() = " + b2.p() + "\tb2.q() = " + b2.q() );
          System.out.println();

          // 'A' is ASCII 65
          System.out.println( "b1.f( 'A' ) = " + b1.f( 'A' ) );
          System.out.println( "b2.f( 'A' ) = " + b2.f( 'A' ) );
          System.out.println();

          System.out.println( "b1.f( 65 ) = " + b1.f( 65 ) );
          System.out.println( "b2.f( 65 ) = " + b2.f( 65 ) );
          System.out.println();
          }
     }
```

## Appendix For Question 6

```
[ 10 ]int a;
^int p;
[ 10 ][ 5 ]int b;

int i = 4;
int x;
^int y;

class List
     begin
          int value;
          ^List next;
     end

[ 10 ]List list;

^List z;

y = a;
y = p;
x = p^;
x = a[ i ];
y = &a[ i ];
y = b[ i ];

z = list[ i ];
z++;
```

Appendices Continued ...

## Commonly used Alpha instructions

### Integer operate instructions

`Opcode $regA, $regB, $regC`

intReg[ regC ] = intReg[ regA ] op intReg[ regB ]

`Opcode $regA, constantB, $regC`

The constant is an 8 bit unsigned constant.
intReg[ regC ] = intReg[ regA ] op constantB

**Arithmetic integer operate instructions**

| | | |
|---|---|---|
| addq | add | + |
| subq | subtract | - |
| mulq | multiply | * |
| divq/divqu | divide, signed/unsigned | / |
| modq/modqu | modulo, signed/unsigned | % |
| s8addq | scaled 8 add | 8*operandA+operandB |

**Shift integer operate instructions**

| | | |
|---|---|---|
| sll | shift left logical | << |
| srl | shift right logical | >>> |
| sra | shift right arithmetic | >> |

**Compare integer operate instructions**

| | | |
|---|---|---|
| cmpeq | compare equal | == |
| cmplt/cmpult | compare less than signed/unsigned | < |
| cmple/cmpule | compare less than or equal signed/unsigned | <= |

**Logical integer operate instructions**

| | | |
|---|---|---|
| and | and | & |
| bic | bit clear | & ~ |
| bis/or | bit set/or | \| |
| eqv/xornot | equivalent/exclusive or not | ^ ~ |
| ornot | or not | \| ~ |
| xor | exclusive or | ^ |

Appendices Continued ...

**Memory instructions**

```
Opcode $regA, displacement($regB)
Opcode $regA, ($regB)
Opcode $regA, constant
```

The displacement or constant is a 16 bit signed constant.

**Load address instruction**

intReg[ regA ] = displacement + intReg[ regB ]

| | |
|---|---|
| lda | load address |

**Load memory instructions**

intReg[ regA ] = Memory[ displacement + intReg[ regB ] ]

| | |
|---|---|
| ldq | load quadword |
| ldl | load longword |
| ldbu | load byte unsigned |

**Store memory instructions**

Memory[ displacement + intReg[ regB ] ] = intReg[ regA ]

| | |
|---|---|
| stq | store quadword |
| stl | store longword |
| stb | store byte |

**Branch instructions**

**Conditional branch instructions**

```
Opcode $regA, destination
```

if ( condition holds for intReg[ regA ] )
    programCounter = destination

| | |
|---|---|
| beq | branch equal |
| bne | branch not equal |
| blt | branch less than |
| ble | branch less than or equal |
| bgt | branch greater than |
| bge | branch greater than or equal |
| blbs | branch low bit set |
| blbc | branch low bit clear |

**Unconditional branch instructions**

```
Opcode destination;
```

programCounter = destination // br

intReg[ ra ] = programCounter // bsr
programCounter = destination

| | |
|---|---|
| br | branch |
| bsr | branch to subroutime |

**Jump instruction**

`Opcode ($regA);`

programCounter = intReg[ regA ] // jmp

intReg[ ra ] = programCounter // jsr
programCounter = intReg[ regA ]

| | |
|---|---|
| jmp | jump |
| jsr | jump to subroutine |

**Return instruction**

programCounter = intReg[ ra ]

| | |
|---|---|
| ret | return |

**Callpal instruction**

`call_pal constant;`

The constant is a 26 bit constant.

| | |
|---|---|
| call_pal | call PALcode |

**Pseudoinstructions**

**Load immediate**

`ldiq $regA, constant`

The constant is a 64 bit constant.
intReg[ regA ] = constant

| | |
|---|---|
| ldiq | load immediate quadword |

**Clear**

`clr $regA`

intReg[ regA ] = 0

| | |
|---|---|
| clr | clear |

**Unary pseudoinstructions**

`Opcode $regB, $regC`

intReg[ regC ] = op intReg[ regB ]

`Opcode constantB, $regC`

The constant is an 8 bit unsigned constant.
intReg[ regC ] = op constantB

| | |
|---|---|
| mov | move |
| negq | negate |

__________________End of Appendices__________________